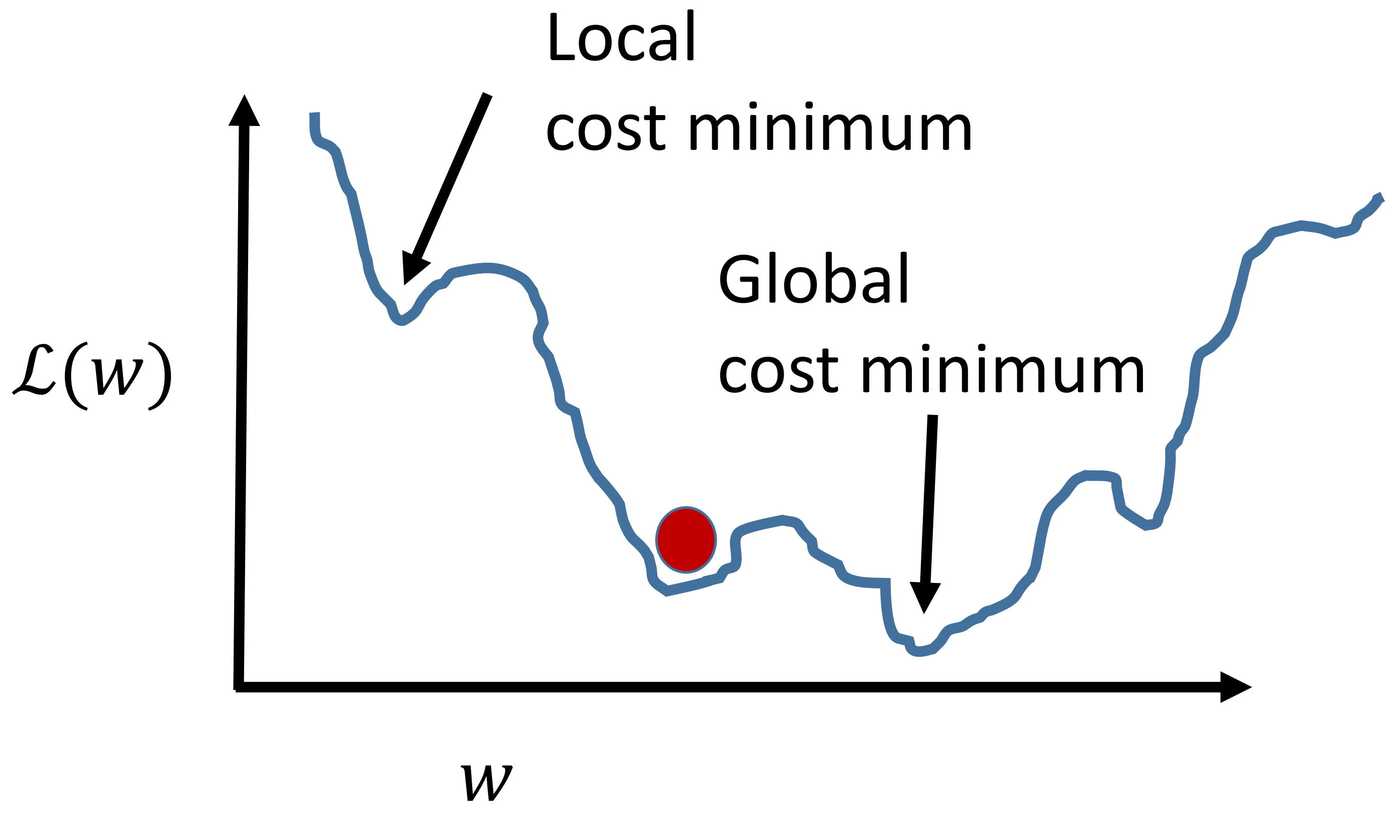
Practical Aspects for Training Deep Networks

Saket Anand

Slide Credits: Vineeth Balasubramanium, IIT Hyd.; Chetan Arora, IIT Delhi

Challenges of Gradient Descent

- Unlike convex objective functions that have a global minimum, non-convex functions as in DL have multiple local minima



Loss Functions

- Mean Squared Error $$\text{loss}(x, y) = (x - y)^2$$
  - Regression problems.
  - The numerical value features are not large.
  - Problem is not very high dimensional

- Smooth L1 loss (Huber loss) Used over the above one $$\text{loss}(x, y) = \begin{cases} 0.5(x - y)^2, & \text{if } |x - y| < 1 \\ |x - y| - 0.5, & \text{otherwise} \end{cases}$$
  - Regression
  - When the features have large values.
  - Well suited for most problems

Weight Initialization

- What do you think?
  - What if we initialized weights with small random numbers?
  - Works okay for small networks, but can lead to non-homogeneous distributions of activations across layers of a network
- To be chosen randomly, **but in such a way that the activation function is in its linear region**
  - Both large and small weights can cause very low gradients
- Assuming inputs to a unit are uncorrelated with variance 1, standard deviation of units weighted sum is: $\sigma_{y_i} = \sqrt{\sum_j w_{ij}^2}$
- Then weights should be randomly drawn from a distribution with mean zero and a standard deviation given by: $\sigma_w = 1/\sqrt{m}$, where m is number of inputs to the unit.

Weight Initialization

Most recommended today:

- Xavier's initialization: $uniform(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." AISTATS 2010

- Caffe implements a simpler version of Xavier's initialization as: $uniform(-\frac{2}{fan_{in}+fan_{out}}, \frac{2}{fan_{in}+fan_{out}})$

- He's initialization: $uniform(-\frac{4}{fan_{in}+fan_{out}}, \frac{4}{fan_{in}+fan_{out}})$
  - He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." ICCV 2015

Data Manipulation Methods

Shuffling Input

- LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 9-48
- Choose examples with maximum information content
  - Shuffle the training set so that successive training examples never (rarely) belong to the same class.
- Present input examples that produce a large error more frequently than examples that produce a small error. Why? **Helps take large steps in the gradient descent**
- Do you see any problems? **What if the data sample is an outlier?**
- Is this relevant for Batch GD?

Relevant for mini-batch but not for complete

Data Augmentation

Methods

- Data jittering (E.g. Distortion and blurring of images)
- Rotations
- Color changes
- Noise injection
- Mirroring

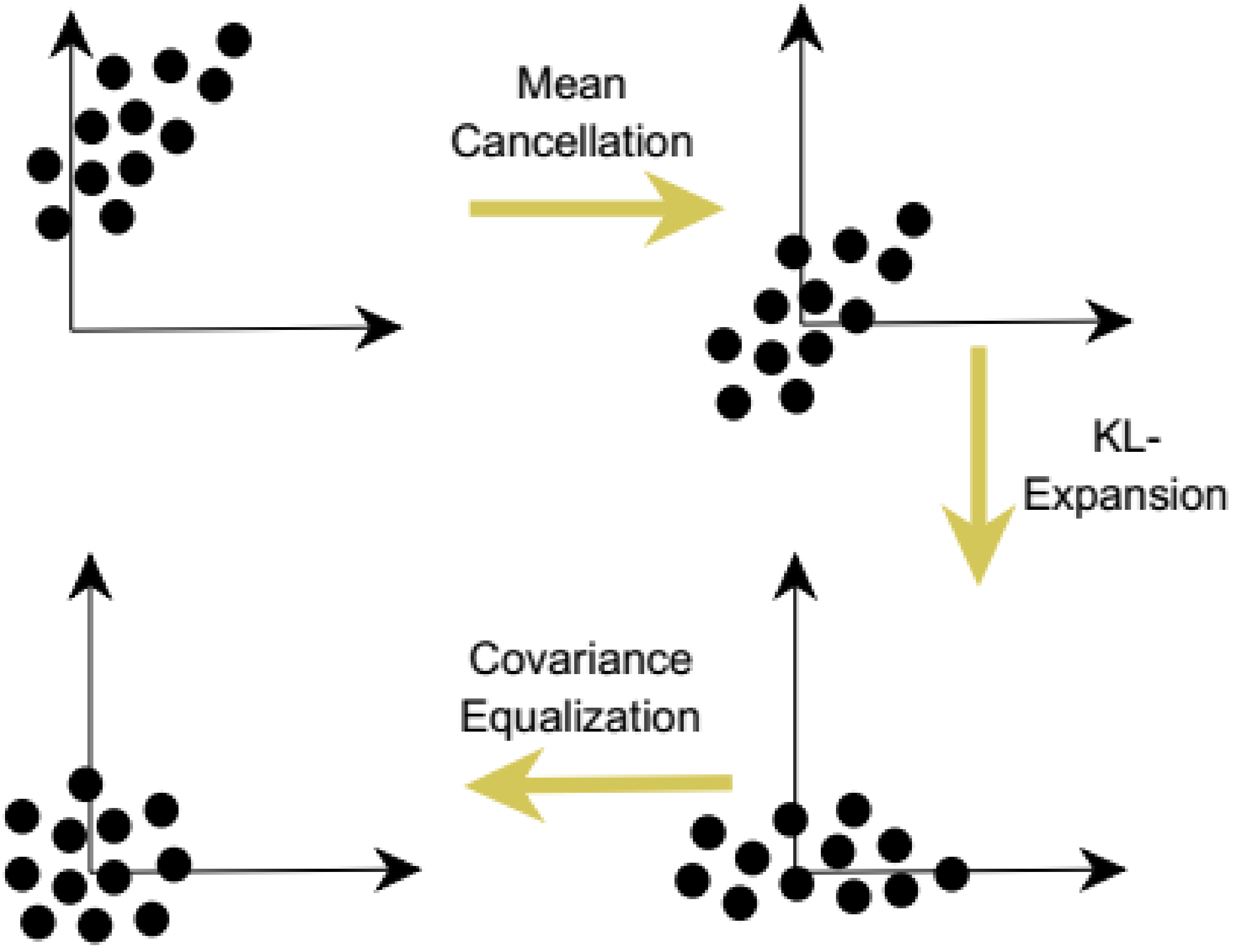
Benefits

- Helps increase data; is useful when training data provided is less (DNNs need large amounts of training data to work!)
- Also acts as a regularizer (by avoiding overfitting to provided data)

Data Transformation

- Normalize/standardize the inputs
    - Convergence is faster if average input over the training set is close to zero.
    - Consider the extreme case when all the inputs are positive
        - Weight update = error at the node * input
        - All weight updates will either increase or decrease ‘together’
        - Weight vector update will be zigzag and inefficient
- Scaled to have the similar covariance - speeds learning.
- Ideally, value of covariance should be matched with output of activation function (e.g. sigmoid)
    - Will ensure that the input to each layer remains zero mean and unit covariance.

Data Transformation



KL Expansion or KLT is similar to PCA and it is often said that optimally compresses the energy. However, KLT is harder to compute than PCA and less popular

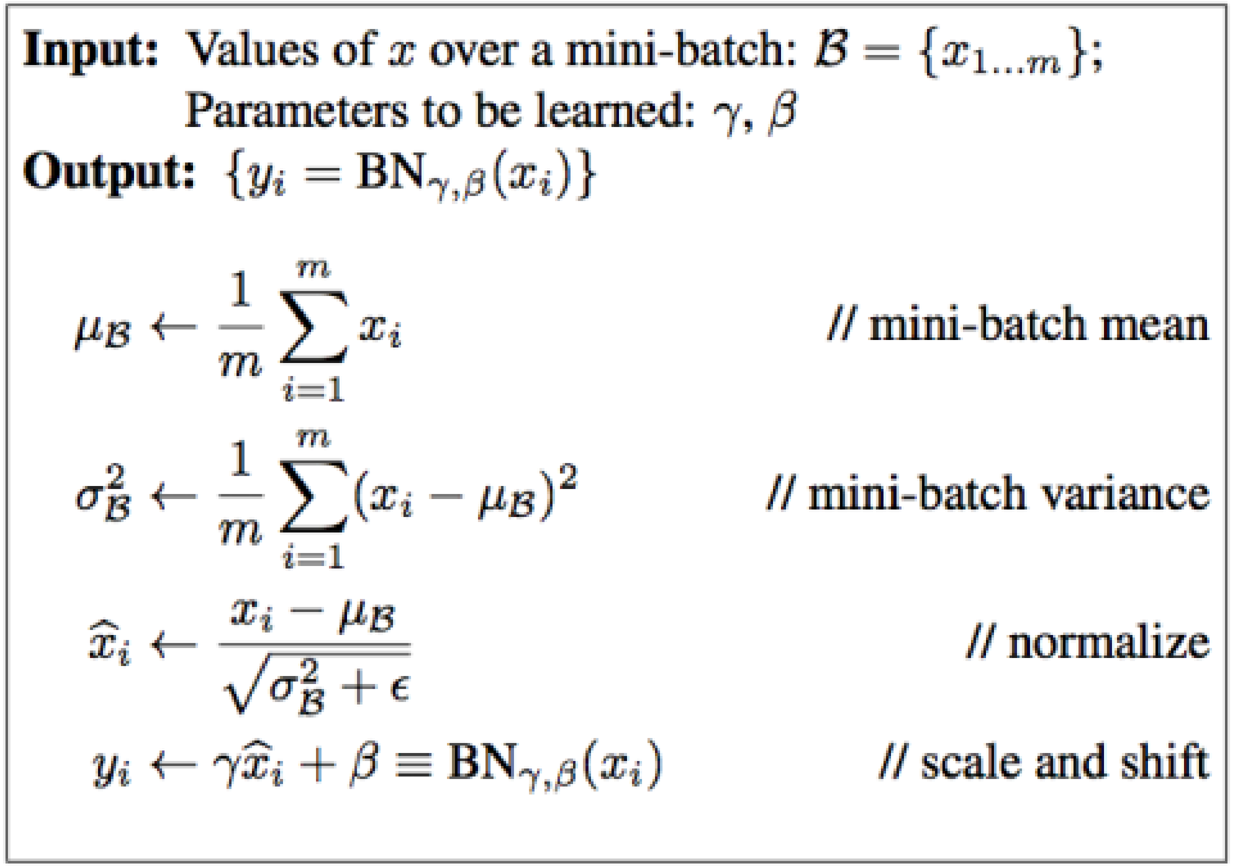
Yann Lecun, Leon Bottou, Genevieve Orr and Klaus-Robert Miller. "Efficient Backprop".

Covariate Shift

- A classifier learned from the source domain not perform well on the target domain if $P_s(X) \neq P_t(X)$
- Becomes a problem when misspecified models are used.
- Intuitively the best model is chosen which minimizes the expected classification error. Therefore, the optimal model performs better in dense regions of X than in sparse regions of X, because the dense regions dominate the average classification error, which is what we want to minimize.
- If the dense regions of X are different in the source and the target domains, the optimal model for the source domain will no longer be optimal for the target domain.

Batch Normalization

- What if this happens in a subnetwork in DL (called internal covariate shift). How to handle?
- Ioffe, Sergey, and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” arXiv preprint 2015
- Whiten every layer’s inputs. Helps obtain a fixed distribution of inputs into each layer



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$    // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$    // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$    // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$    // scale and shift
```

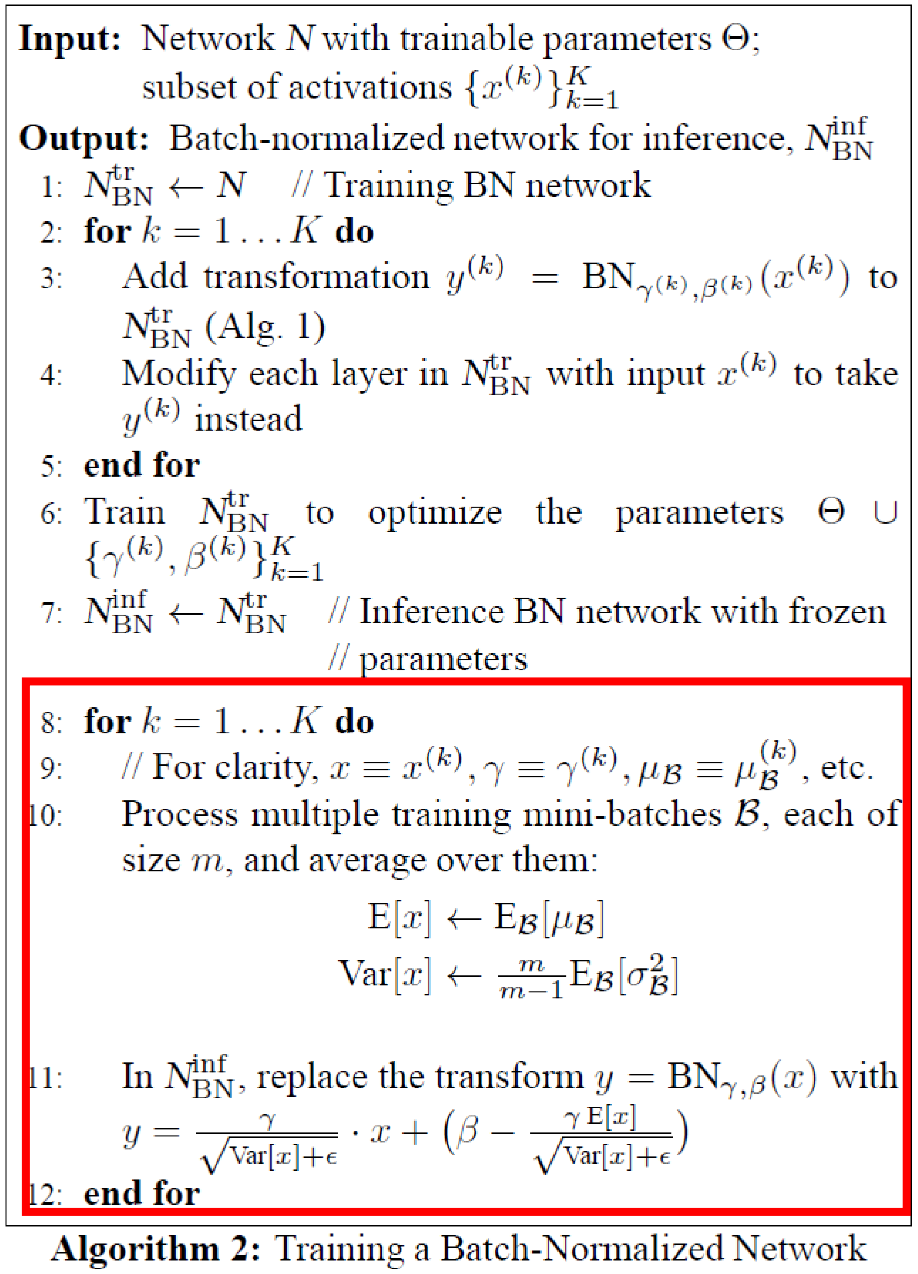
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

- BN layer usually inserted before non-linearity layer (after FC or convolutional layer)
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization too

- How do we handle test time?
- Evaluate a mini-batch at a time?

Breaking News: It’s not about covariate shift!
https://arxiv.org/pdf/1805.11604.pdf

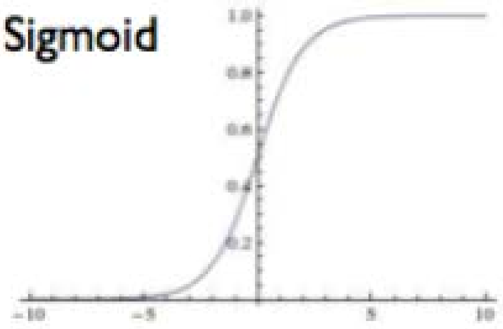


```
Input: Network $N$ with trainable parameters $\Theta$;
       subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\mathrm{BN}}^{\mathrm{inf}}$
 1: $N_{\mathrm{BN}}^{\mathrm{tr}} \leftarrow N$    // Training BN network
 2: for $k = 1 \dots K$ do
 3:     Add transformation $y^{(k)} = \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to
        $N_{\mathrm{BN}}^{\mathrm{tr}}$ (Alg. 1)
 4:     Modify each layer in $N_{\mathrm{BN}}^{\mathrm{tr}}$ with input $x^{(k)}$ to take
        $y^{(k)}$ instead
 5: end for
 6: Train $N_{\mathrm{BN}}^{\mathrm{tr}}$ to optimize the parameters $\Theta \cup$
    $\{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
 7: $N_{\mathrm{BN}}^{\mathrm{inf}} \leftarrow N_{\mathrm{BN}}^{\mathrm{tr}}$    // Inference BN network with frozen
                     // parameters
 8: for $k = 1 \dots K$ do
 9:     // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10:     Process multiple training mini-batches $\mathcal{B}$, each of
        size $m$, and average over them:
            $\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$
            $\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$
11:     In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with
        $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$
12: end for
```

Algorithm 2: Training a Batch-Normalized Network

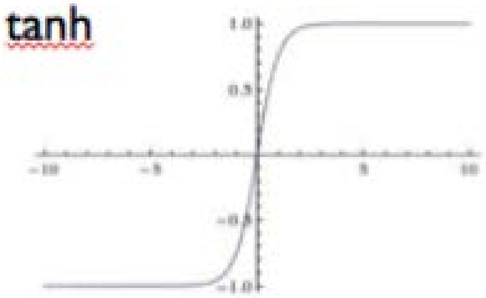
Activation Function

Sigmoid



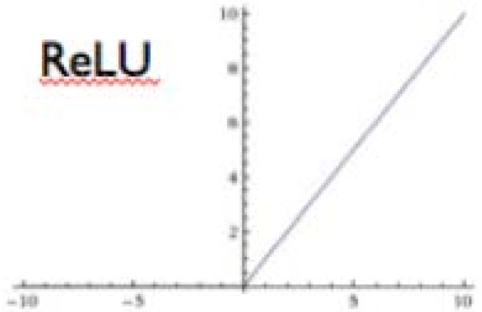
$$y = \frac{1}{1+e^{-x}}$$

tanh



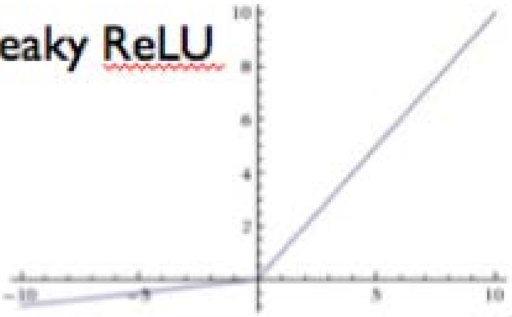
$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

ReLU



$$y = max(0, x)$$

Leaky ReLU



$$y = \begin{cases} x & \text{if } x < 0 \\ 0.01x & \text{if } otherwise \end{cases}$$

maxout

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

Activation Function

- ReLUs the default option today
- The dying ReLU problem → the leaky ReLU
- Found to accelerate convergence of SGD compared to sigmoid/tanh functions (a factor of 6) in AlexNet
- Compared to tanh/sigmoid neurons that involve expensive operations (e.g. exponentials), can be implemented by simply thresholding a matrix of activations at zero.
- MaxOut ! a generalization of ReLUs

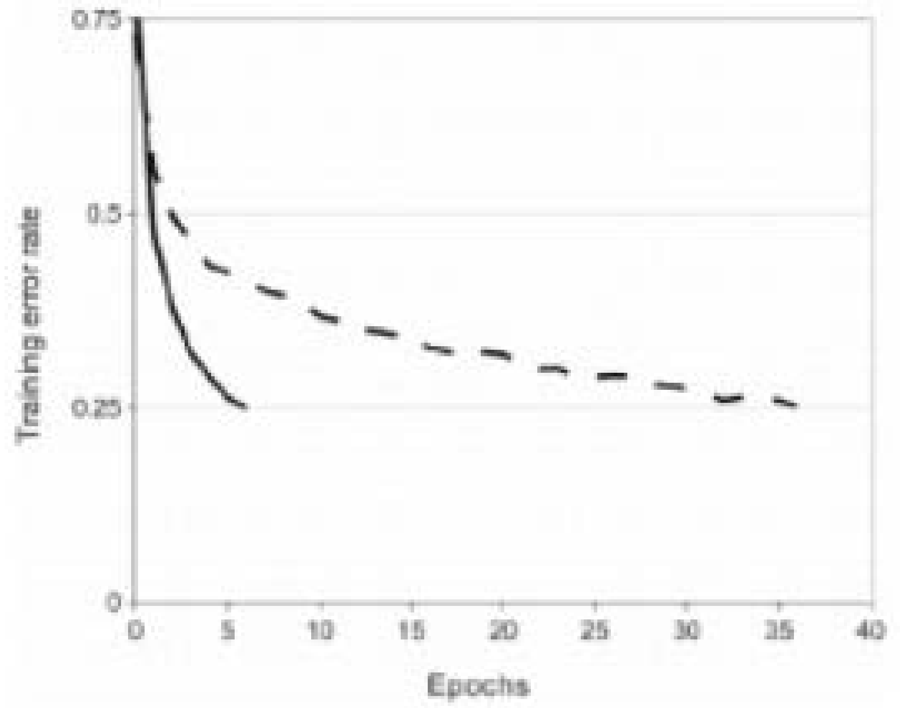


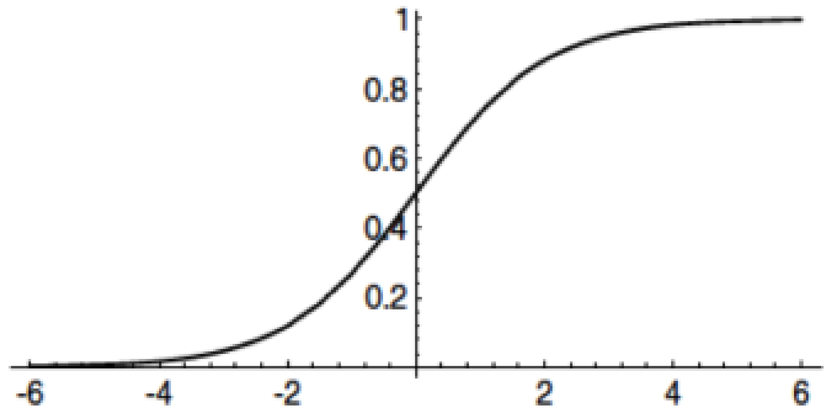
Figure 1: A four-layer convolutional neural network with ReLUs (**solid line**) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

Activation Function: Which one to choose?

As advised by Fei-Fei Li (Stanford) in her course “CNNs for Visual Recognition”

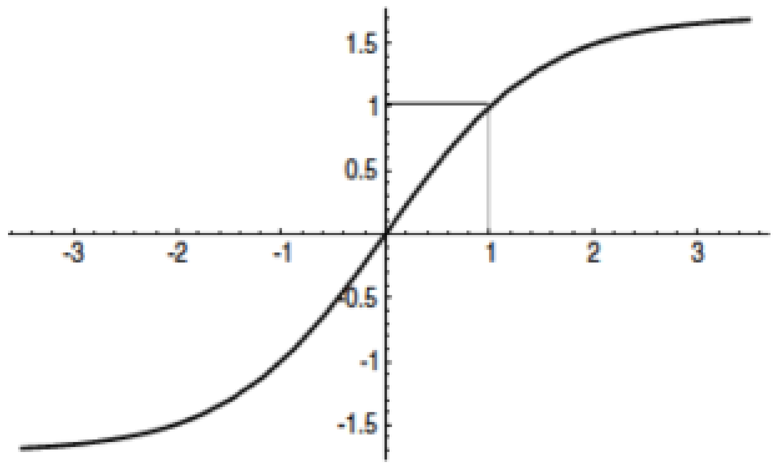
- Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of “dead” units in a network.
- If this concerns you, give Leaky ReLU or Maxout a try.
- Never use sigmoid.
- Try tanh, but expect it to work worse than ReLU/Maxout

Which of the sigmoids will you prefer?



Logistic Function

$$f(x) = \frac{1}{1 + e^{-x}}$$



Tanh Function

$$f(x) = 1.7159 \tanh(\frac{2}{3}x)$$

Tanh: Has zero mean and unit variance

Optimization Methods

Challenges of Gradient Descent

- The non-identifiability problem results in multiple equivalent local minima. Not problematic though, since cost function value is the same
- Local minima with significant cost value differences are the problem
- Some prior work says that it is not important to find a true global minimum, rather than to find a point in parameter space that has low but not minimal cost. (Choromanska, Mathieu & LeCun, "The Loss Surface of Multilayer Nets", AISTATS'2015)

How to know if you are in one of the local minima?

- Plot the norm of the gradient over time. If the norm is very small, it is likely to be a local minimum (or a critical point).

Exploding/Vanishing Gradient

- Deeper the network, gradients vanish quickly, thereby slowing the rate of change in initial layers
- Problem accentuated in long-term RNNs
- Exploding gradients happen when the individual layer gradients are much higher than 1, for instance

Slow Convergence

Given the issues:

- Cost surface is often non-quadratic, non-convex, high-dimensional
- Potentially, many minima and flat regions

No guarantee that:

- Network will converge to a good solution
- Convergence is swift
- Convergence occurs at all

Other Challenges

- Ill-conditioning
- Inexact gradients
- Choosing learning rate, other parameters/hyperparameters

How to address these challenges

Algorithmic Approaches

- Batch GD, SGD, Mini-batch SGD
- Momentum, Nesterov Momentum
- Adagrad, Adadelta, RMSProp, Adam
- Advanced Optimization Methods

How to address these challenges

Practical Tricks

- Regularization Methods (including Dropout)
- Data Manipulation Methods
- Parameter Choices/Initialization Methods (Activation Functions, Loss Functions, Weights)

Batch GD, Stochastic GD and Mini-Batch SGD

- **Batch GD:** Update the parameters after the gradients are computed for the entire training set
- **Stochastic GD:** Randomly shuffle the training set, and update the parameters after gradients are computed for each training example
- **Mini-Batch Stochastic GD:** Update the parameters after gradients are computed for a randomly drawn mini-batch of training examples (this is the default option today)

Batch GD, Stochastic GD and Mini-Batch SGD

Advantages of SGD

- Usually much faster than batch learning. Why? Redundancy in batch learning
- Often results in better solutions. Why? Noise can help!
- Can be used for tracking changes. Why and how? Some systems can change over time.

Issues with SGD

- Noise in SGD weight updates - can lead to no convergence!
- Can be controlled using learning rate
- Equivalent to use of "mini-batches" in SGD (Start with a small batch size and increase size as training proceeds)

Batch GD, Stochastic GD and Mini-Batch SGD

Advantages of Batch GD

- Conditions of convergence are well understood.
- Many acceleration techniques (e.g. conjugate gradient) only operate in batch learning.
- Theoretical analysis of the weight dynamics and convergence rates are simpler.

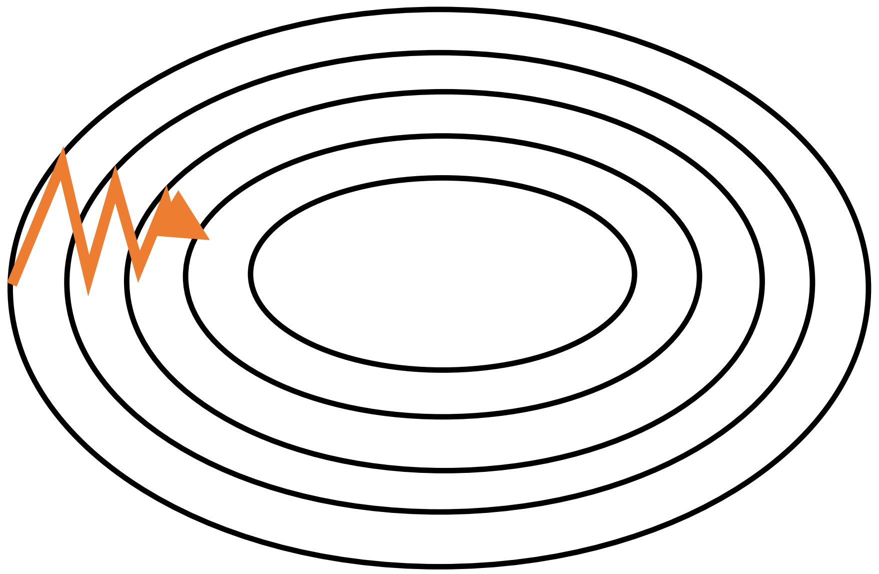
Mini-batch SGD is the most commonly used method

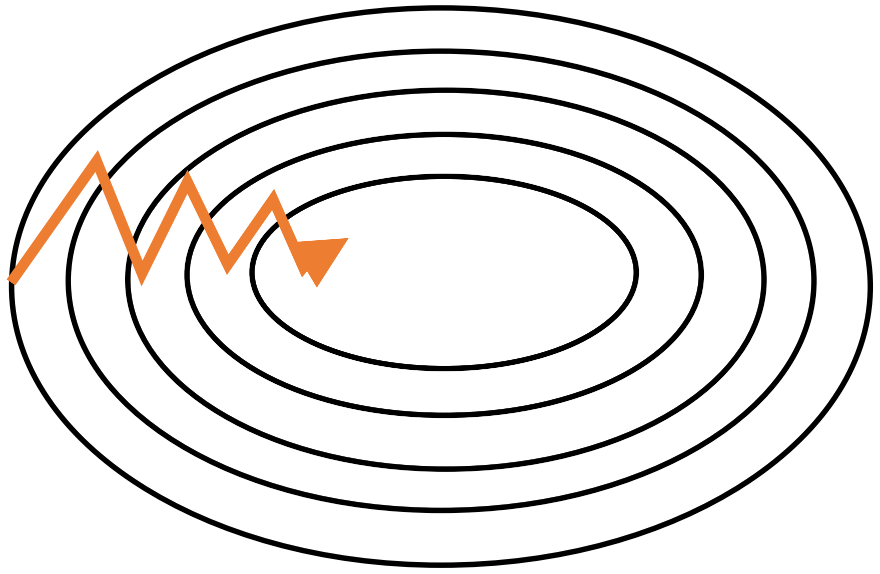
Momentum

- Weight update given by:

Momentum Term

$$\Delta\theta_{t+1} = \alpha\nabla_\theta\mathcal{L}\left(\theta_t; x^{(i)}, y^{(i)}\right) + \overset{\Downarrow}{\gamma\Delta\theta_t}$$


Without Momentum


With Momentum

Momentum

- Weight update given by:

$$\Delta\theta_{t+1} = \alpha\nabla_\theta\mathcal{L}\left(\theta_t; x^{(i)}, y^{(i)}\right) + \overset{\substack{\text{Momentum Term}\\ \Downarrow}}{\boxed{\gamma\Delta\theta_t}}$$

- Can increase speed when the cost surface is highly non-spherical
- Damps step sizes along directions of high curvature, yielding a larger effective learning rate along the directions of low curvature
- Larger the γ, more the previous gradients affect the current step
- Generally γ is set to 0.5 until initial learning stabilizes and then increased to 0.9 or higher

Nestorov Momentum

- Weight update given by:

$$\Delta\theta_{t+1} = \alpha \nabla_\theta \mathcal{L}\left(\theta_t + \overset{\Downarrow}{\boxed{\gamma\Delta\theta_t}}; x^{(i)}, y^{(i)}\right) + \gamma\Delta\theta_t$$

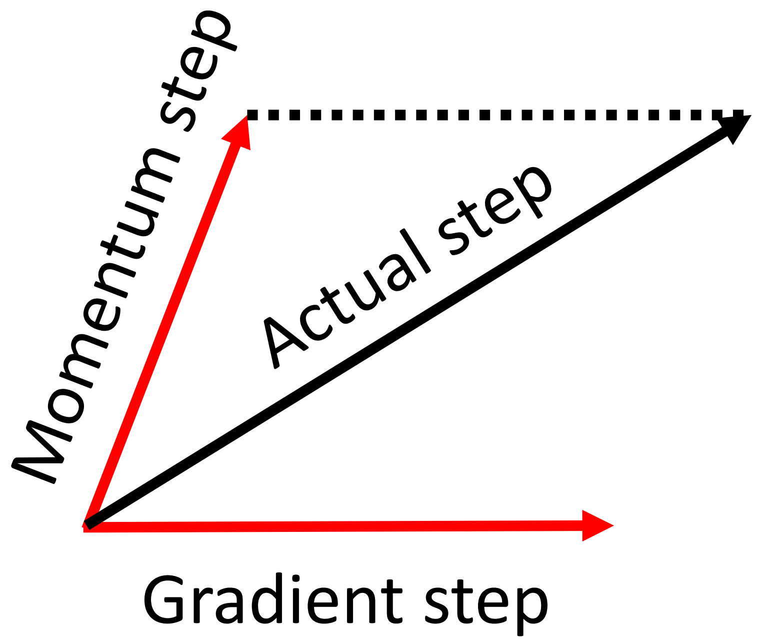
- What’s the difference?
- Where the gradient is evaluated is different

Nestorov Momentum

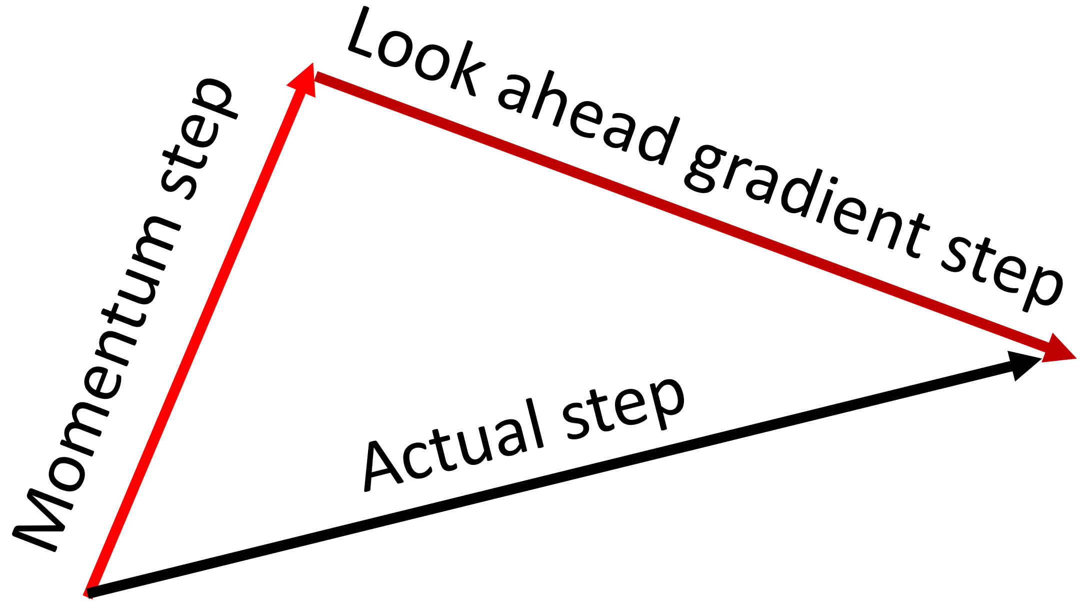
- Weight update given by:

$$\Delta\theta_{t+1} = \alpha \nabla_\theta \mathcal{L}\left(\theta_t + \gamma\Delta\theta_t; x^{(i)}, y^{(i)}\right) + \gamma\Delta\theta_t$$

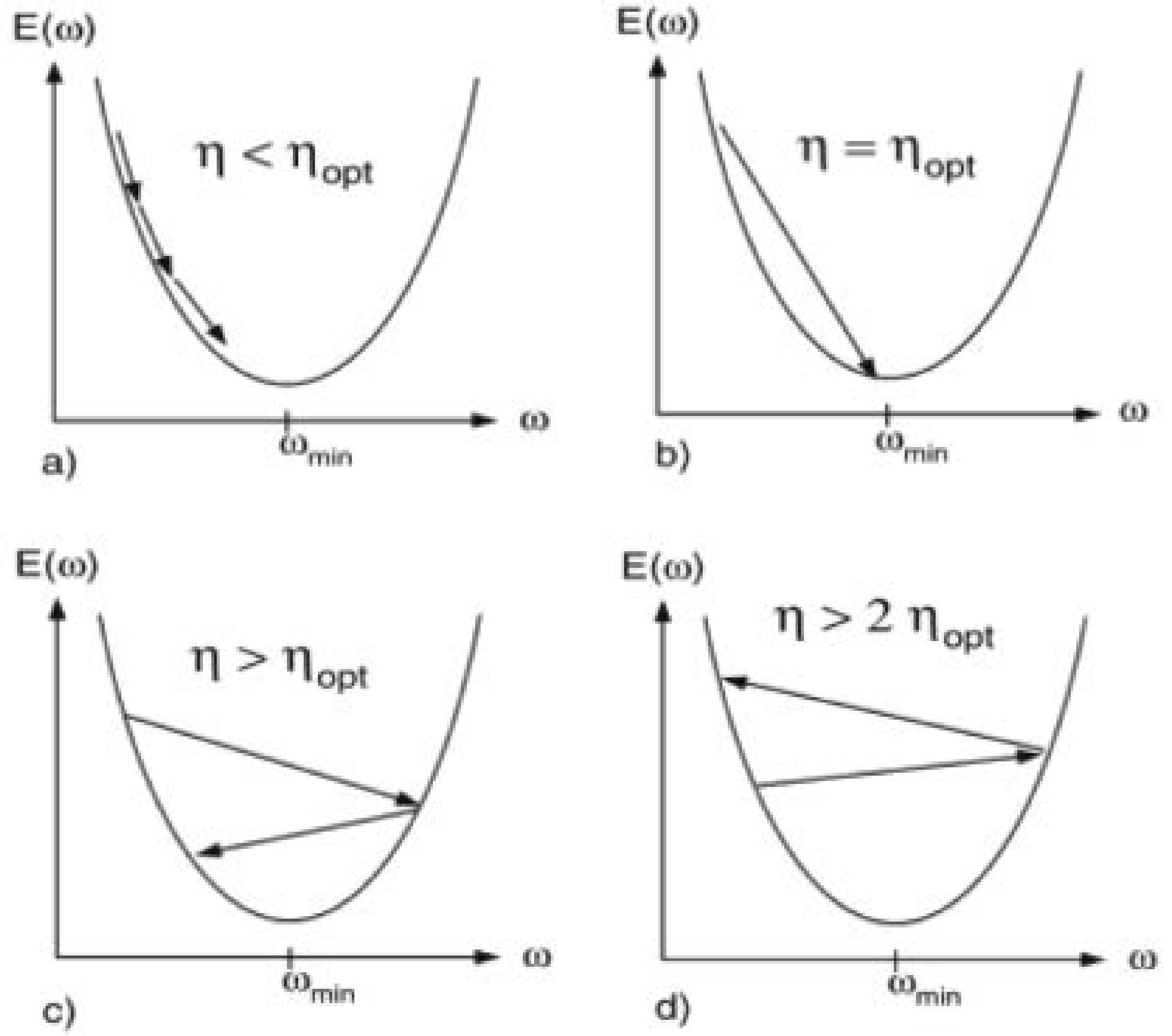
Momentum Update



Nestorov Momentum Update

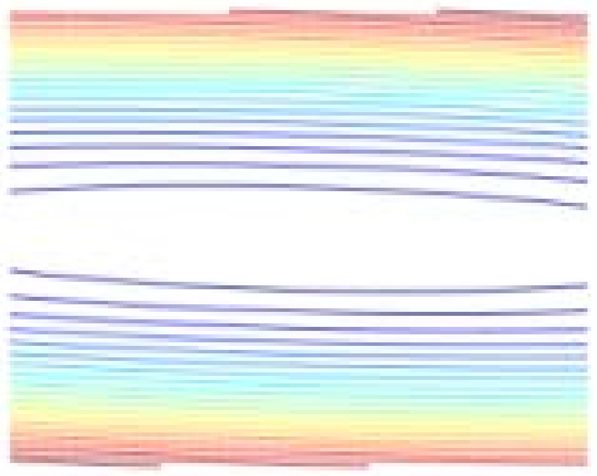


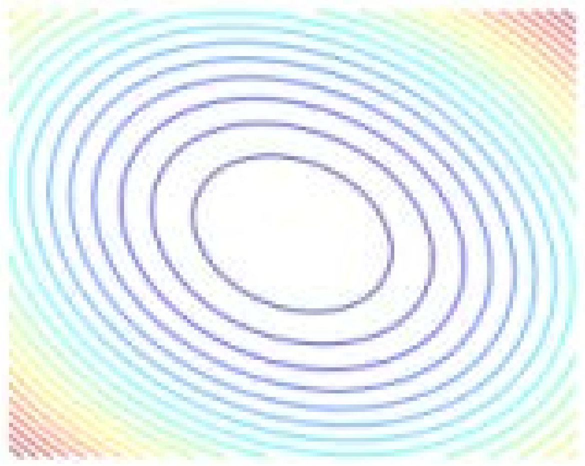
What is Optimal Learning Rate α



Adagrad

- Calculates a different learning rate for each feature
- Sparse features should have higher learning rate


Hard


Nice

RMSProp

Require: Global learning rate α, Decay rate ρ, Minibatch size m, Initial weights θ_t, Small constant δ, usually 10^{-6}, for numerical stability

```
1.  Initialize accumulation variable $r = 0$
2.  while stopping criterion not met do
3.      Sample a minibatch of m exampl the training set
4.      Compute gradient esti
5.      Accumulate s gradient $r = \rho r + (1-\rho)(\nabla_\theta \mathcal{L} \odot \nabla_\theta \mathcal{L})$
6.      Co pdate $\Delta\theta_{t+1} = \frac{\alpha}{\delta + \sqrt{r}} \odot \nabla_\theta \mathcal{L}$ (division and square root
          computed elementwise)
8.      Apply update $\theta_{t+1} = \theta_t - \Delta\theta_{t+1}$
9.  end while
```

Intuition: Change learning rate as a function of accumulated squared gradient

ADAM

Update biased first moment estimate $s = \rho_1 s + (1 - \rho_1)\nabla_\theta \mathcal{L}$
Update biased second moment estimate $r = \rho_2 r + (1 - \rho_2)(\nabla_\theta \mathcal{L} \odot \nabla_\theta \mathcal{L})$
Correct bias in first moment $\tilde{s} = \frac{s}{1-\rho_1^t}$
Correct bias in second moment $\tilde{r} = \frac{r}{1-\rho_2^t}$
Compute update $\Delta\theta_{t+1} = \alpha \frac{\tilde{s}}{\delta+\sqrt{\tilde{r}}}$ (division and square root computed element-wise). $\theta_{t+1} = \theta_t - \Delta\theta_{t+1}$

What's the intuition?

- Similar to RMSProp with momentum
- Uses the idea of momentum, as well as having a different learning rate for each dimension (which is automatically adjusted, as in Adagrad, or RMSProp)

Ignorance is bliss: Now, which one to choose?

A nice tutorial and visualization:

http://sebastianruder.com/optimizing-gradient-descent/

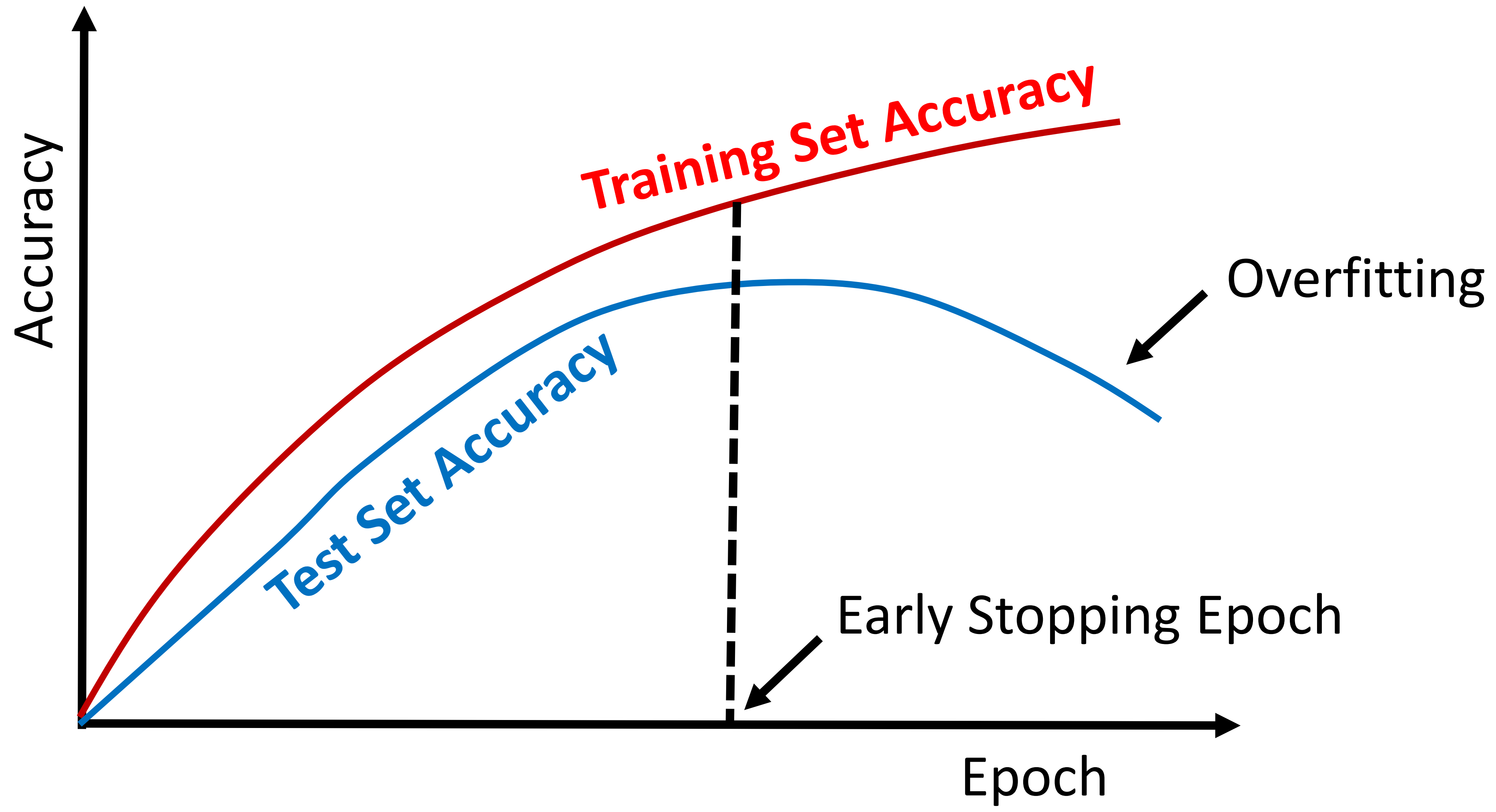
- If input data is sparse, adaptive learning-rate methods may be best.
  - Additional benefit: No need to tune learning rate
- Learning rates diminish fast in Adagrad ! RMSProp addresses this issue
- Adam adds bias-correction and momentum to RMSprop
- RMSprop, Adam are similar algorithms! Bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser
- Adam might be the best overall choice (May not be always true!)

Regularization Methods

Difference between ML and Optimization

- Both try to optimize/minimize empirical risk
- Difference is in **generalization**
- In mainstream optimization, minimizing empirical risk is itself the goal; whereas in deep learning, minimizing risk so as to minimize a generalizable out-of-sample performance measure is the goal
- Minimizing empirical risk as the only objective can lead to overfitting.
- Avoiding overfitting is the regularization

Early Stopping



Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs ! Bad idea: can’t assume one-size-fits-all approach

Error-change criterion:

  - Stop when error isn’t dropping over a window of, say, 10 epochs
  - Train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)

Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs ! Bad idea: can't assume one-size-fits-all approach

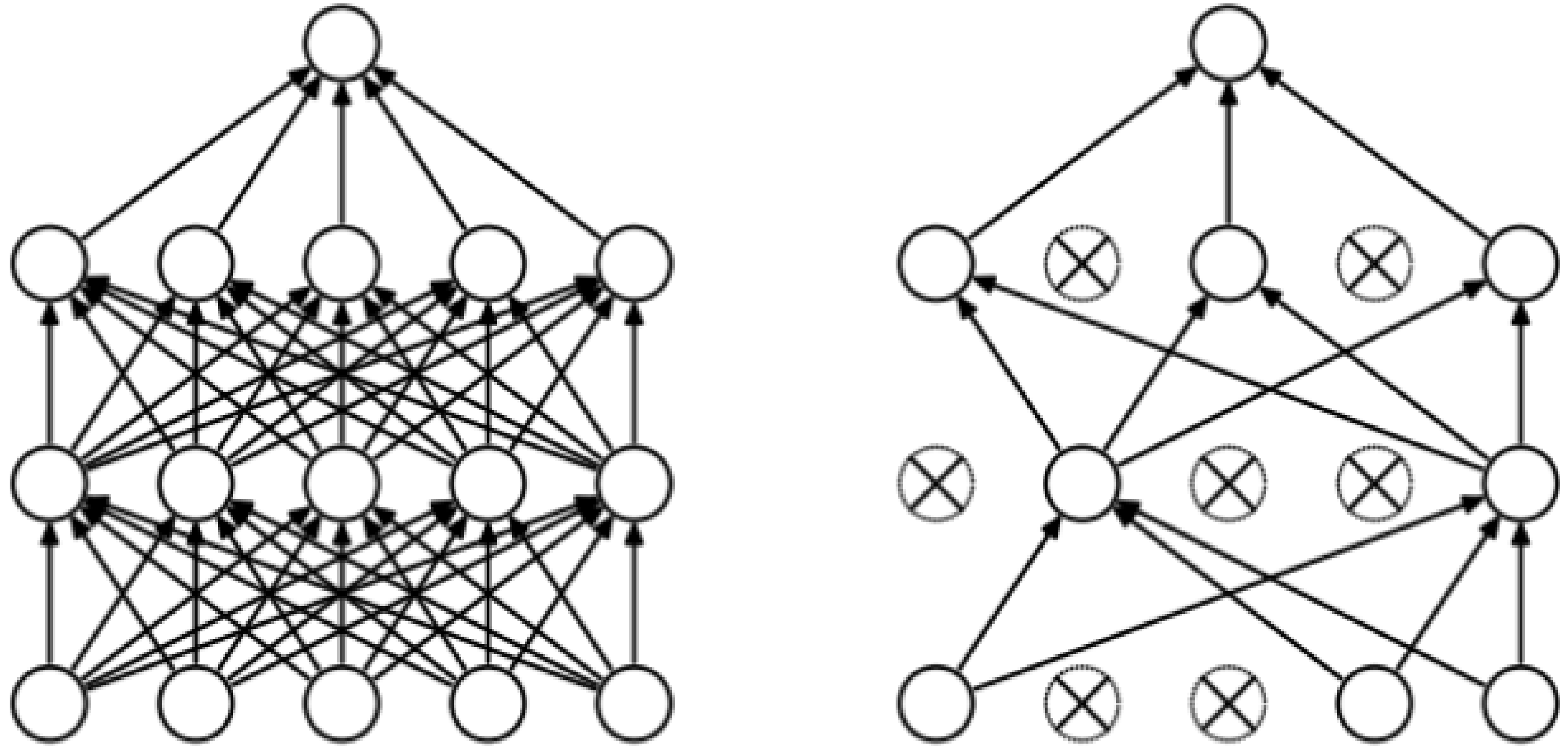
Weight-change criterion:

- Compare weights at epochs t-10 and t and test: $\max_i \left| w_i^t - w_i^{t-10} \right| < \rho$
- Don't base on length of overall weight change vector
- Possibly express as a percentage of the norm of the weight vector (relative error)

Dropout

- Another standard approach to regularization in ML: Model Averaging
- DropOut ! a very interesting way to perform model averaging in deep learning
- Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014
- **Training Phase:** For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, *p*, of nodes (and corresponding activations)
- **Test Phase:** Use all activations, but reduce them by a factor *p* (to account for the missing activations during training)

Dropout


(a) Standard Neural Net (b) After applying dropout.

Dropout

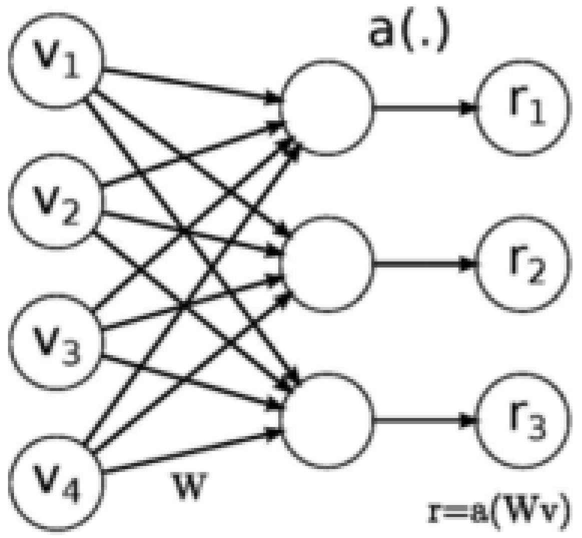
- With H hidden units, each of which can be dropped, we have 2^H possible models

- Each of the 2^{H-1} models that include hidden unit h must share the same weights for the units.
  - Serves as a form of regularization
  - Makes the models cooperate

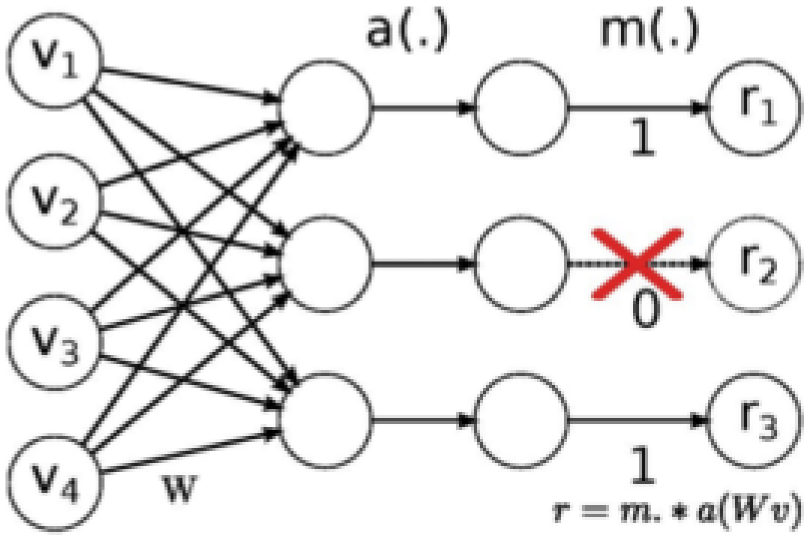
- Including all hidden units at test with a scaling of 0.5 is equivalent to computing the geometric mean of all 2^H models

DropConnect

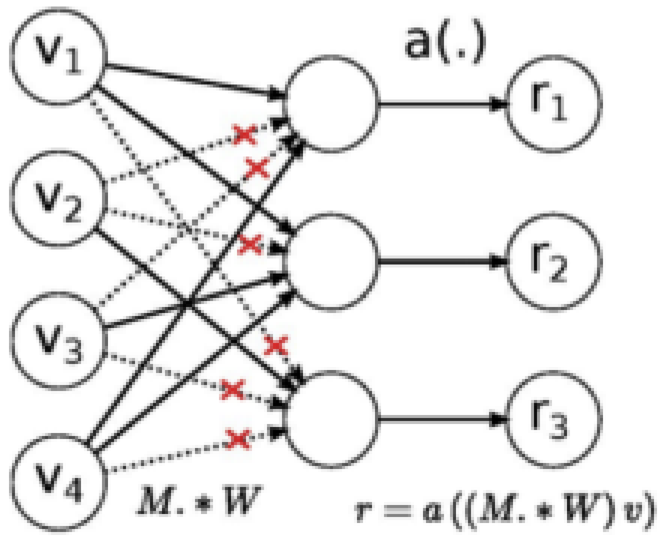
Wan, Li, et al. “Regularization of neural networks using dropconnect”, ICML 2013



No Drop Network



DropOut Network



DropConnect Network

Noise in Data, Label and Gradient

Using noise is another form of regularization; has shown some impressive results recently. Could be:

- **Data Noise**
  - Has been there for a while: add noise to data while training
  - Minimization of sum-of-squares error with zero-mean Gaussian noise(added to training data) is equivalent to minimization of sum-of-squares error without noise with an added regularized term 12
  - Very similar to data augmentation
- **Label Noise**
- **Gradient Noise**

Regularization through Label Noise

- Xie, Lingxi, et al. “DisturbLabel: Regularizing CNN on the Loss Layer”, CVPR 2016

- Disturb each training sample with the probability α

- For each disturbed sample, label is randomly drawn from a uniform distribution over $\{1,2,\dots,C\}$ regardless of the true label

Regularization through Gradient Noise

- Neelakantan, Arvind, et al. "Adding gradient noise improves learning for very deep networks." arXiv preprint arXiv:1511.06807 (2015)

- Simple idea: add noise to gradient

$$g_t \leftarrow g_t + N(0, \sigma^2)$$

- Annealed Gaussian noise by decaying the variance

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma} \sigma_{t-1}^2$$

- Showed significant improvement in performance

Conclusion

- Some standard choices for training deep networks: SGD + Nesterov momentum, SGD with Adagrad/RMSProp/Adam
- ReLUs, Leaky ReLUs and MaxOut are the best bets for activation functions
- Batch Normalization layers are here to stay (at least, for now)
- DropOut is an excellent regularizer
- Data Augmentation is a must in vision applications
- Weight Initialization is very important while training a new network